Approximate pattern matching with k-mismatches in packed text

Emanuele Giaquinta[1], Szymon Grabowski[2], and Kimmo Fredriksson[3]

[1] Department of Computer Science, University of Helsinki, Finland
emanuele.giaquinta@cs.helsinki.fi
[2] Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, 90–924 Łódź, Poland sgrabow@kis.p.lodz.pl
[3] School of Computing, University of Eastern Finland, P.O.B. 1627, FI-70211 Kuopio, Finland kimmo.fredriksson@uef.fi

Abstract. Given strings P of length m and T of length n over an alphabet of size σ, the string matching with k-mismatches problem is to find the positions of all the substrings in T that are at Hamming distance at most k from P. If T can be read only one character at the time the best known bounds are $O(n\sqrt{k \log k})$ and $O(n + n\sqrt{k/w} \log k)$ in the word-RAM model with word length w. In the RAM models (including AC^0 and word-RAM) it is possible to read up to $\Theta(w/\log \sigma)$ characters in constant time if the characters of T are encoded using $\lceil \log \sigma \rceil$ bits. The only solution for k-mismatches in packed text works in $O((n \log \sigma / \log n) \lceil m \log(k + \log n / \log \sigma)/w \rceil + n^{\varepsilon})$ time, for any $\varepsilon > 0$. We present an algorithm that runs in time $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor}(1 + \log \min(k, \sigma) \log m / \log \sigma))$ in the AC^0 model if $m = O(w/\log \sigma)$ and T is given packed. We also describe a simpler variant that runs in time $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor} \log \min(m, \log w / \log \sigma))$ in the word-RAM model. The algorithms improve the existing bound for $w \gg \log n$. Based on the introduced technique, we present algorithms for several other approximate matching problems.

1 Introduction

The string matching problem consists in reporting all the occurrences of a pattern P of length m in a text T of length n, both strings over a common alphabet. The occurrences may be exact or approximate according to a specified matching model. For most matching problems, all the characters from the text and the pattern need to be read at least once in the worst case; hence, if they are read one at a time, the worst-case lower bound is $\Omega(n)$. Interestingly, for some standard problems (e.g., exact pattern matching) it is possible to achieve a sublinear search time, for short patterns, even in the worst case, if the word-RAM computational model is assumed and the text is *packed*. In a packed encoding, the characters of a string are stored adjacently in memory and each character is encoded using $\log \sigma$ bits[4], where σ is the alphabet size. A single machine word,

[4] Throughout the paper, all logarithms are in base 2. W.l.o.g. we also assume that σ is a power of two.

of size $w \geq \log n$ bits, thus contains up to $\alpha = \lfloor w/\log\sigma \rfloor$ characters. For most of the following considerations we assume to have a short pattern, i.e, one which fits in a word ($m \leq \alpha$).

For this setting and the exact string matching problem, several sublinear-time algorithms have been given in recent years [12, 7, 5, 6, 8].

In this paper we study the string matching with k-mismatches problem in the packed scenario. This problem is to find the positions of all the substrings in T that are at Hamming distance at most k from P, i.e., that match P with at most k mismatches. For this problem, the best known bounds in the worst-case are $O(n\sqrt{k \log k})$ of the algorithm by Amir et al. [2] and $O(n + n\sqrt{k/w} \log k)$ of its implementation based on word-level parallelism [13]. One classical result in the word-RAM model that is also practical is the Shift-Add algorithm [4]. The best worst-case bound of this algorithm, based on the Matryoshka counters technique [14], is $O(n\lceil m/w \rceil)$.

In [12] Fredriksson presented a Shift-Add variant, based on the super-alphabet technique, that works in $O((n \log\sigma/\log n)\lceil m \log(k+\log n/\log\sigma)/w\rceil + n^{\varepsilon})$ time, for any $\varepsilon > 0$. To our knowledge, this is the only solution for the k-mismatches problem that works on packed text and that achieves sublinear time complexity when m and k are sufficiently small.

In this work, we present an algorithm for the k-mismatches problem that runs in time $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor}(1+\log\min(k,\sigma)\log m/\log\sigma))$ in the AC^0 model for $m \leq \alpha$ if T is given packed. We also describe a simpler variant that runs in time $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor}\log\min(m, \log w/\log\sigma))$ in the word-RAM model. In particular, it achieves sublinear worst-case time when $m\log\sigma\log\min(m, \log w/\log\sigma) = o(w)$. Note that for $w = \Theta(\log n)$ Fredriksson's solution is better, but our algorithm dominates if $w \gg \log n$, which becomes a fairly standard assumption nowadays.

2 Basic notions and definitions

Let $\Sigma = \{0, 1, \ldots, \sigma-1\}$ denote an integer alphabet and Σ^m the set of all possible sequences of length m over Σ. $S[i], i \geq 0$, denotes the $(i+1)$-th character of string S, and $S[i \ldots j]$ its substring between the $(i+1)$-st and the $(j+1)$-st characters (inclusive).

The k-mismatches problem consists in, given a pattern (string) P of length m and a text (string) T of length n, reporting all the positions $0 \leq j \leq n-m$ such that $|\{0 \leq h < m \ : \ T[j+h] \neq P[h]\}| \leq k$, i.e., such that the Hamming distance between P and the substring $T[j \ldots j+m-1]$ is at most k.

The word-RAM model is assumed, with machine word size $w = \Omega(\log n)$. We use some bitwise operations following the standard notation as in C language: &, |, $\hat{}$, $\sim$, <<, >> for `and`, `or`, `xor`, `not`, `left shift` and `right shift`, respectively.

3 The algorithm

We start the presentation with a simple idea, which is then extended and modified in some ways. We define an (f)-word as a machine word logically divided into $\lfloor w/f \rfloor$ fields of f bits. The most significant bit in a field is called the *top* bit. We also define, for a given field size f, the (constant) word $V_f = 10^{f-1} \dots 10^{f-1}$. Consider two $(\log \sigma)$-words A and B, each containing a packed string of length m in its $m \log \sigma$ least significant bits (i.e., each field of $\log \sigma$ bits encodes a character). The higher bits in both words, if any, are all 0s (where there can be no misunderstanding, we will silently assume such convention about used and non-used bits from now on). We perform the xor operation of A and B and the number of non-zero fields in the result is exactly the Hamming distance between the two strings. To have this calculation efficient, we make use of two primitives on a machine word:

- *find non-zero fields* ($\mathsf{fnf}(A, f)$): given a (f)-word A, return a word in which a bit is set to 1 if it is the top bit of a non-zero field in A and to 0 otherwise.
- *sideways addition* ($\mathsf{sa}(A)$): given a word A, return the number of bits set in A.

How to implement the first primitive in $O(1)$ time was presented in [8, Sect. 4], but we will give a simpler procedure. The second primitive is a well-known bitwise operation, also known as popcount. The folklore method[5] to compute it has $O(\log \log w)$ time complexity. The procedure to compute the Hamming distance of A and B can thus be implemented with the following operations:

1. $A' \leftarrow A \wedge B$
2. $A' \leftarrow \mathsf{fnf}(A', \log \sigma)$
3. **return** $\mathsf{sa}(A')$

Using this method, we can obtain an algorithm for the string matching with k-mismatches problem that runs in $O(n\lceil m \log \sigma / w \rceil \log \log w)$ time. Note that the resulting algorithm is also practical and compares favorably with the classical Shift-Add algorithm [4] for small alphabets and large k, although it is less flexible (no support for classes of characters). It is also worth noting that recent processors include a POPCNT instruction to compute the sideways addition of a word, so the $\log \log w$ term disappears in practice.

We now explain how to implement the fnf primitive, in three simple steps and in constant time. The input is a $(\log \sigma)$-word A.

1. $W \leftarrow A \mathbin{\&} \sim V_{\log \sigma}$
2. $A' \leftarrow V_{\log \sigma} - W$
3. $A' \leftarrow (\sim A' \mid A) \mathbin{\&} V_{\log \sigma}$

[5] `http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel`

We now define two variants of the sideways addition and another known primitive on words that we will use in a refined variant of our technique.

interleaved blockwise sideways addition ($\mathsf{ibsa}(A, f, b)$): Given an (f)-word A, with $f \leq \log \sigma$ and such that only the top bit of each field may be set, and a power of two b, return the word $A'_{\log b}$, where

$$A'_j = \begin{cases} A'_{j-1} + (A'_{j-1} << (2^{j-1} \times \log \sigma)) & \text{if } j > 0 \\ A >> (f-1) & \text{otherwise} \end{cases}$$

This operation computes the sums of all the sequences of b f-bit fields spaced by $\log \sigma$ bits and is a variant of the parallel prefix-sum operation described in [15]. Each sum is stored into the last field of the corresponding sequence. Since $f \geq \log(b+1)$ does not necessarily hold, the top bits of all the fields are masked out before each addition and restored afterwards. In this way, if a sum is $\geq 2^{f-1}$, its encoded value is $\geq 2^{f-1}$ but the exact value is undetermined. This operation can be implemented in time $O(\log b)$.

blockwise sideways addition ($\mathsf{bsa}(A, f, b)$): Given an (f)-word A, such that only the top bit of each field may be set, and a power of two b, return a word in which each block of bf bits contains the number of bits set in the corresponding b fields in A.

This operation can be implemented in time $O(\log \min(b, \log w/f))$ using the following method. We assume that the word size w is a power of two. Let r be the smallest power of two greater than or equal to $\log(w+1)/f$. The first step consists in computing a word logically divided into fields of $\min(b, r)f$ bits, such that each field contains the number of bits set in the corresponding $\min(b, r)$ fields in the original word. This widening operation can be performed in $\log \min(b, r) = O(\log \min(b, \log w/f))$ steps using simple bitwise operations.

Since both r and b are a power of two, each block spans an integral number of fields of $\min(b, r)f$ bits. Observe that there can be at most w bits set in a word, so rf bits are enough to encode the total number of bits. If $b \leq r$, then since b is a power of two after the last widening step we have a word divided into fields of bf bits, each one containing the desired number of ones. Otherwise, if $b > r$, we use a multiplication with the mask $0^{rf-1}1 \ldots 0^{rf-1}1$ to store into each field the sum of the previous fields including itself. This corresponds to computing the prefix sum of the sequence of numbers given by the fields. It is not hard to see that, after the multiplication, the number of bits in a block is equal to the last field of the block minus the last field of the previous block. This operation can be implemented in parallel for all the blocks with a shift and a subtraction.

Observe that the claimed bound holds only if we assume that multiplication is $O(1)$, which is true in the word-RAM model but not in the AC^0 model. In the AC^0 model the bsa operation can be performed in $O(\log b)$ time by applying $O(\log b)$ widening steps.

parallel minima (maxima) [17] (pmin (pmax)): Given two (b)-words A and B, return a word in which a bit is set to 1 if it is the top bit of a block in A whose

value is smaller (larger) than or equal to the value of the corresponding block in B and to 0 otherwise. pvmin (pvmax) is similar, but instead of setting the top bits, returns the actual minimum (maximum) value pair-wise for each field.

These operations can be implemented in constant time, as demonstrated by the following code (pmin)

1. $T_A \leftarrow A \;\&\; V_b$
2. $T_B \leftarrow B \;\&\; V_b$
3. $A' \leftarrow A \;\&\; \sim V_b$
4. $A'' \leftarrow (B \mid V_b) - A'$
5. $H_1 \leftarrow \sim A'' \;\&\; \sim T_A \;\&\; T_B$
6. $H_2 \leftarrow A'' \;\&\; \sim T_A \;\&\; \sim T_B$
7. $H_3 \leftarrow A'' \;\&\; \sim T_A \;\&\; T_B$
8. $H_4 \leftarrow A'' \;\&\; T_A \;\&\; T_B$
9. $A''' \leftarrow (H_1 \mid H_2 \mid H_3 \mid H_4) \;\&\; V_b$

We now show how to apply the described ideas in an (improved) algorithm for the k-mismatches problem on packed text for short patterns ($m \leq \alpha$). Our method exploits a general technique [16] to increase the parallelism in string matching algorithms based on word-level parallelism. We first present a solution in the AC^0 model, and then describe a simpler variant that is better in the word-RAM model. Let $\bar{m}$ be the smallest power of two greater than or equal to m and let $\ell = \lfloor w/(\bar{m}\log\sigma)\rfloor$. We first preprocess the pattern P to create a word A with ℓ copies of P of length $\bar{m}\log\sigma$ starting from the least significant bit. The last $\bar{m} - m$ fields of each copy are set to zero. We also maintain a word H, initialized to 0, that will be used during the searching phase. Let B_i be the word containing the packed encoding of the substring $T[j \ldots j + \ell m - 1]$, where $j = \ell\lfloor i/m\rfloor m + i \bmod m$, with $\bar{m} - m$ zero (padding) fields every m fields. Note that the word B_i, for $i > 0$, can be computed incrementally from B_{i-1} and the packed text in constant time, using simple bitwise operations. Let $\bar{k}$ be the smallest power of two greater than k. Our search algorithm performs the following main steps, for each $0 \leq i < n/\ell$:

1. $A' \leftarrow A \wedge B_i$
2. $A' \leftarrow \mathsf{fnf}(A', \log\sigma)$
3. $H \leftarrow (H << f) \mid A'$
4. **if** $i > 0$ and $i \bmod \lfloor \log\sigma/f \rfloor = 0$
5. $\quad M \leftarrow \mathsf{pmin}(\mathsf{ibsa}(H, f, \bar{m}), K, f)$
6. $\quad$ **report**(M)
7. $\quad H \leftarrow 0$

where $f = \min(\log\bar{k} + 1, \log\sigma)$ and K is a word with $\ell\lfloor\log\sigma/f\rfloor$ copies of the integer k spaced by $f(\bar{m} - 1)$ bits. At each iteration, our algorithm processes ℓ substrings of T in parallel using the technique to compute the Hamming distance of two words described before. However, we report the occurrences every $\lfloor\log\sigma/f\rfloor$ iterations, so as to reduce the overhead due to counting the number

of mismatches for each substring when $\log k = o(\log \sigma)$. To this end, we compact the fields in the word $\mathsf{fnf}(A \wedge B_i, \log \sigma)$ into fields of size f in the word H. If $i > 0$ and $i \bmod \lfloor \log \sigma / f \rfloor = 0$, i.e., every $\ell \lfloor \log \sigma / f \rfloor$ processed substrings, we report the occurrences as follows. First, observe that the word H contains $\ell \bar{m} \lfloor \log \sigma / f \rfloor$ fields of f bits, encoding the mismatches for the substrings of T of length m corresponding to the words B_{i-j}, for $j = 0, \ldots, \lfloor \log \sigma / f \rfloor - 1$. More precisely, the l-th sequence of $\mathsf{fnf}(A \wedge B_{i-j}, \log \sigma)$ spans the fields $s, s + \lfloor \log \sigma / f \rfloor, \ldots, s + \lfloor \log \sigma / f \rfloor (\bar{m} - 1)$, where $s = jf + (l-1)\bar{m} \lfloor \log \sigma / f \rfloor$, for $l = 1, \ldots, \ell$. Using a suitable algorithm, i.e, the ibsa operation, we compute a word such that the last field of each sequence has value equal to the number of bits set (mismatches) in all the fields of the sequence if the number of mismatches is less than $\bar{k}$ and to a value $\geq \bar{k}$ otherwise. Then, to find all the occurrences with at most k mismatches we use the pmin operation with the word K to identify the blocks with a bit count less than or equal to k. Finally, to iterate over all the occurrences we use the well-known bitwise operation that computes the position of the highest bit set in a word. Observe that this operation is in AC^0 and takes constant time [3]. The time complexity of this algorithm is $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor}(1 + \log \min(k, \sigma) \log m / \log \sigma))$. It obtains the $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor})$ bound, corresponding to no overhead for the bitwise operations, if $\log \min(k, \sigma) \log m = O(\log \sigma)$.

We now present the simpler variant in the word-RAM model. Let r be the smallest power of two greater than or equal to $\log(w+1)/\log \sigma$. The algorithm performs the following main steps, for each $0 \leq i < n/\ell$:

1. $A' \leftarrow A \wedge B_i$
2. $A' \leftarrow \mathsf{fnf}(A', \log \sigma)$
3. $M \leftarrow \mathsf{pmin}(\mathsf{bsa}(A', \log \sigma, \bar{m}), K, f)$
4. **report**(M)

where in this case $f = \min(r, \bar{m}) \log \sigma$ and K contains ℓ copies of k spaced by $\bar{m} \log \sigma - f$ bits. The main difference in this algorithm is that we do not defer the reporting of occurrences. Rather, at each iteration we use the bsa operation, which is cheap in this model, to compute into fields of f bits the number of bits set (mismatches) in each block of $\bar{m}$ fields of A'. Observe that in this setting bsa has $O(\log \min(m, \log w / \log \sigma))$ time complexity. Hence, our algorithm has $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor} \log \min(m, \log w / \log \sigma))$ time complexity, and it obtains the $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor})$ bound for $\log \sigma = \Omega(\log w)$ or constant m.

Finally, we give a variant useful for two extreme cases: either k or $m-k$ is very small. More precisely, it is competitive when $k = o(\log \min(k, \sigma) \log m / \log \sigma)$ or $m - k = o(\log \min(k, \sigma) \log m / \log \sigma)$. It uses only AC^0 instructions. In this variant, each pattern copy in A has $p = 1 + m \log \sigma$ associated bits. The most significant (extra) bit is a sentinel that will signal that there are more than k mismatches, as will be shown shortly. The remaining $m \log \sigma$ bits encode the pattern as usual. The idea is to parallelize the well-known sideways addition

implementation in which the least significant bit set is cleared in a loop[6]. To this end, we perform the following procedure:

1. $A' \leftarrow A \wedge B_i$
2. $A' \leftarrow \mathsf{fnf}(A', \log \sigma)$
3. **for** $i \leftarrow 1$ **to** $k+1$
4. $\quad A' \leftarrow A' \mid V_p$
5. $\quad A' \leftarrow A' \,\&\, (A' - (V_p >> (p-1)))$
6. $M \leftarrow (A' \,\&\, V_p) \wedge V_p$
7. **report**(M)

After the last operation we obtain set bits only in the positions corresponding to pattern copies with at most k mismatches. That is, in those cases even the last subtraction does not clear the sentinel bit. The complexity of the described operation is $O(k)$. The time complexity of this algorithm is $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor} k)$. A twin solution handles the case of small $m - k$, which basically consists in using the same method on the bitwise complement of the top bits of A'.

4 Applications

The presented technique can be used for several other string matching problems. We show how to adapt it for particular models in the following subsections.

4.1 Matching with k-mismatches and wildcards

Assume that the alphabet Σ, of size σ, contains a wildcard symbol, i.e., a special symbol that matches any other symbol of the alphabet. Hence we count the mismatches only for regular characters [9]. We consider the case in which wildcards may occur in both the pattern and the text.

In the preprocessing we create, in $O(\log(w/(m \log \sigma)))$ time using only AC^0 instructions, or even in $O(1)$ time in the word-RAM, two $(\log \sigma)$-words W_P and H_T. The word W_P contains $\ell \bar{m}$ fields. A field in W_P has the top bit set to 0 if the field corresponds to a wildcard position in the word A, to 1 otherwise. All the other bits are set to 0. The word H_T contains $\ell \bar{m}$ copies of the wildcard symbol.

At each iteration i of the searching phase, we compute the word $W_T = \mathsf{fnf}(B_i \wedge H_T, \log \sigma)$. Analogously to W_P, a field in W_T has the top bit set to 0 if the field corresponds to a wildcard position in B, to 1 otherwise. All the other bits are set to 0.

Then, we `and` the result of operation 2 of the algorithm with $W_P \,\&\, W_T$ (i.e., $A' \leftarrow A' \,\&\, (W_P \,\&\, W_T)$), which effectively means that there can be no mismatch in a position where either a pattern or text wildcard occurs. The rest of the procedure is unchanged. The overall time complexity is also unchanged, apart from the initial (negligible) $O(\log(w/(m \log \sigma)))$ term.

[6] `http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetKernighan`

4.2 δ-matching with k-mismatches and (δ, γ)-matching

In δ-matching [10] any two characters t and p are defined to match iff $|t-p| \leq \delta$. Combined with Hamming distance, a text position j matches the pattern if $|\{0 \leq i < m \ : \ |T[j+i] - P[i]| > \delta_i\}| \leq k$. Note that we can allow δ_i to be different for each pattern position i, while usually in δ-matching the allowed error δ is the same for each character. This also yields an alternate solution to matching with wildcards in the pattern, by simply using $\delta_i = \sigma - 1$ for pattern positions i corresponding to wildcards, and $\delta_i = 0$ elsewhere.

In the preprocessing phase we compute $D'[j] = \delta_j$ and construct its packed representation D, a $(\log \sigma)$-word holding ℓ copies of D'. Let W be a $(\log \sigma)$-word with top bits set in fields corresponding to pattern characters. Then at iteration i of the searching phase compute

1. $X \leftarrow \mathsf{pvmax}(A, B_i, \log \sigma) - \mathsf{pvmin}(A, B_i, \log \sigma)$
2. $A' \leftarrow \mathsf{pmin}(X, D, \log \sigma) \wedge W$

The result is that in every field of A' the top bit is 1 iff the corresponding pattern and text characters do not δ-match. The rest of the algorithm is as before, only the steps 1–2 of either of the main algorithms (for AC^0 and word-RAM models) are effectively replaced with the two steps above. The time complexities remain the same.

If we are interested in the (more conventional) exact δ-matching variant (i.e. assume that $k = 0$), we can improve the time to $O(\frac{n}{\lfloor w/(m/\log \sigma) \rfloor})$ using the following algorithm:

1. $X \leftarrow \mathsf{pvmax}(A, B_i, \log \sigma) - \mathsf{pvmin}(A, B_i, \log \sigma)$
2. $A' \leftarrow \mathsf{pmin}(X, D, \log \sigma) \wedge W$
3. $M \leftarrow \mathsf{fnf}(A', m \log \sigma) \wedge V_{m \log \sigma}$
4. **report**(M)

The fnf operation interprets the word A' as $(m \log \sigma)$-word, and sets the top bit to 1 for each pattern copy where even one character did not δ-match, and the xor operation then inverts those top bits, giving bit 1 for pattern copies that did δ-match in all character positions.

We note that a close relative to δ-matching is less-than matching, where characters p and t match if $p \leq t$. This model has applications in other pattern matching problems, see e.g. [1]. Less-than matching problem can be solved in our methods easily in the same way as δ-matching, that is, the first two lines are simply replaced with $A' \leftarrow \mathsf{pmin}(A, B_i, \log \sigma) \wedge W$. It is also possible to solve δ-matching by combining less-than and greater-than matching.

In γ-matching we sum the absolute differences between character pairs, and limit this sum with a parameter γ, i.e. a text position j γ-matches the pattern if $\sum_{0 \leq i < m} |T[j+i] - P[i]| \leq \gamma$. If both δ and γ conditions must hold, we speak of (δ, γ)-matching. There are many algorithms devoted to this model, see e.g. [11]. Thus in order to solve γ-matching, we need to sum the fields of X and compare against γ (note that we need to check also the δ condition). If we do not use field compaction and deferred reporting of occurrences, this is easiest to do the

same way as the first widening phase of bsa operation, i.e. by simply shifting and adding the fields in parallel in $O(\log m)$ time, giving $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor} \log m)$ total time in AC^0. This result can be improved in both AC^0 and word-RAM models. We first define one more operation:

interleave two words ($\mathsf{interleave}(A, B, S, f)$): given two ($f$)-words A and B, interleave the fields according to (f)-word S. That is, return a word Z where a field is selected from A if the corresponding field in S is zero, and from B otherwise. This can be implemented in $O(1)$ time as follows:

1. $S \leftarrow \mathsf{fnf}(S, f)$
2. $S \leftarrow (S - (S >> (f-1))) \mid S$
3. $Z \leftarrow A \mathbin{\&} \sim S \mid B \mathbin{\&} S$

Consider first the AC^0 model. We use an approach similar to the one used in the k-mismatches case, the only difference is that we need more bits to represent the accumulated sums. That is, we replace $\bar{k}$ with $\bar{\gamma} = 2^{\lceil \log(\gamma+1) \rceil}$ (the smallest power of two greater than γ), and K with a word G, containing $\ell \lfloor \log \sigma / f \rfloor$ copies of the integer γ, where $f = \min(\log \bar{\gamma} + 1, \log \sigma)$. The ibsa operation then works correctly, i.e. accumulates the absolute differences without overflowing the sums, provided that all characters δ-match, as otherwise the corresponding absolute difference may be $\sigma - 1$, which in turn can be larger than $\bar{\gamma}$. To this end, we take care that we do not compute the exact sum if even one character pair does not δ-match. Instead, in such cases we assume that all the differences are δ, to ensure that ibsa works correctly and that the text position will not be reported. This works because in γ-matching it always holds that $m\delta > \gamma$, as otherwise the γ condition does not prune anything. The complete pseudo code follows.

1. $X \leftarrow \mathsf{pvmax}(A, B_i, \log \sigma) - \mathsf{pvmin}(A, B_i, \log \sigma)$
2. $A' \leftarrow \mathsf{pmin}(X, D, \log \sigma) \mathbin{\wedge} W$
3. $Z \leftarrow \mathsf{fnf}(A', \bar{m} \log \sigma) \mathbin{\wedge} V_{\bar{m} \log \sigma}$
4. $X' \leftarrow \mathsf{interleave}(X, D, Z, \bar{m} \log \sigma)$
5. $H \leftarrow (H << f) \mid X'$
6. **if** $i > 0$ and $i \bmod \lfloor \log \sigma / f \rfloor = 0$
7. $\quad M \leftarrow \mathsf{pmin}(\mathsf{ibsa}(H, f, \bar{m}), G, f)$
8. $\quad \mathbf{report}(M)$
9. $\quad H \leftarrow 0$

The total time is $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor}(1 + \log m \log \gamma / \log \sigma))$, This becomes $O(\frac{n}{\lfloor w/(m \log \sigma) \rfloor})$ for $\log m \log \gamma = O(\log \sigma)$.

In word-RAM the algorithm is again more simple. We start as in the AC^0 case to compute the word X', and then use bsa and pmin to accumulate the absolute differences and compare against the threshold value γ in parallel. The only thing remaining is to adjust the parameters so as not to cause overflows in bsa. To this end, we use $r = \lceil \log(w\delta + 1) \rceil / \log \sigma$ and $f = \min(r, \bar{m}) \log \sigma$, and the algorithm is

1. $X \leftarrow \mathsf{pvmax}(A, B_i, \log\sigma) - \mathsf{pvmin}(A, B_i, \log\sigma)$
2. $A' \leftarrow \mathsf{pmin}(X, D, \log\sigma) \wedge W$
3. $Z \leftarrow \mathsf{fnf}(A', \bar{m}\log\sigma) \wedge V_{\bar{m}\log\sigma}$
4. $X' \leftarrow \mathsf{interleave}(X, D, Z, \bar{m}\log\sigma)$
5. $M \leftarrow \mathsf{pmin}(\mathsf{bsa}(X', \log\sigma, \bar{m}), G, f)$
6. $\mathbf{report}(M)$

where the word G contains ℓ copies of the integer γ this time, spaced by $\bar{m}\log\sigma - f$ bits. The time complexity is $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor}\log\min(m, \log(w\delta)/\log\sigma))$ which again obtains $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor})$ time for $\log\sigma = \Omega(\log(w\delta))$ or constant m.

We can also combine the two models, δ-matching with k-mismatches and (δ,γ)-matching, to obtain (δ,k,γ)-matching. In this model we limit the number of characters not δ-matching by k, and the accumulated sum of the absolute differences by γ. The basic idea is to compute two match vectors, M_δ and M_γ, take their bitwise and as $M \leftarrow M_\delta \,\&\, M_\gamma$ and then report the occurrences with respect to M. The vector M_δ can be computed as was already shown. To compute M_γ we just skip the interleave operation to take the absolute differences raw without saturating them with δ. In AC^0 we can use the basic algorithm to compute M_γ in $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor}\log m)$ time, which dominates the total time. However, since the sums need more bits now, there is a significant overhead in the case of the word-RAM algorithm and of the improved AC^0 algorithm. In particular, in the case of the word-RAM algorithm, the time complexity becomes $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor}\log\min(m, \log(w\sigma)/\log\sigma))$, i.e., slightly worse. Instead, in the case of the improved AC^0 algorithm, the overhead makes the algorithm useless. However, we can still manage to obtain $O(\frac{n}{\lfloor w/(m\log\sigma)\rfloor}(1 + \log m\log\gamma/\log\sigma))$ time by modifying the model so that we accumulate the absolute differences only on δ-matching character positions. This can be easily done with the tools already presented, namely using the interleave operation to set non-δ-matching character positions to 0 in word X'.

5 Conclusion

We presented a novel technique for approximate pattern matching with k-mismatches when the text is given in packed form. Assuming the pattern is short enough, it is possible to achieve a sublinear search time, if several pattern copies are matched against different (but adjacent) text substrings at the same time. We described variants of our simple method in the AC^0 and word-RAM models and also considered the case when the number k of allowed errors is small. Moreover, we showed how to adapt our algorithms to other matching models, including approximate matching with wildcard (don't-care) symbols, δ-matching with k-mismatches and (δ,γ)-matching.

References

1. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. In *FOCS*, pages 144–153. IEEE Computer Society, 1997.

2. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.
3. A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theor. Comput. Sci.*, 215(1-2):337–344, 1999.
4. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
5. D. Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Discrete Algorithms*, 14:91–106, 2012.
6. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann. Optimal packed string matching. In S. Chakraborty and A. Kumar, editors, *FSTTCS*, volume 13 of *LIPIcs*, pages 423–432. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
7. P. Bille. Fast searching in packed strings. *J. Discrete Algorithms*, 9(1):49–56, 2011.
8. D. Breslauer, L. Gasieniec, and R. Grossi. Constant-time word-size string matching. In J. Kärkkäinen and J. Stoye, editors, *CPM*, volume 7354 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 2012.
9. R. Clifford, K. Efremenko, E. Porat, and A. Rothschild. k-mismatch with don't cares. In L. Arge, M. Hoffmann, and E. Welzl, editors, *ESA*, volume 4698 of *Lecture Notes in Computer Science*, pages 151–162. Springer, 2007.
10. M. Crochemore, C. S. Iliopoulos, and T. Lecroq. Occurrence and substring heuristics for δ-matching. *Fundam. Inform.*, 56(1-2):1–21, 2003.
11. M. Crochemore, C. S. Iliopoulos, G. Navarro, Y. J. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching and suffix automata. *J. Discrete Algorithms*, 3(2-4):198–214, 2005.
12. K. Fredriksson. Faster string matching with super-alphabets. In A. H. F. Laender and A. L. Oliveira, editors, *SPIRE*, volume 2476 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2002.
13. K. Fredriksson and S. Grabowski. Exploiting word-level parallelism for fast convolutions and their applications in approximate string matching. *Eur. J. Comb.*, 34(1):38–51, 2013.
14. S. Grabowski and K. Fredriksson. Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time. *Inf. Process. Lett.*, 105(5):182–187, 2008.
15. W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.
16. H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM Journal of Experimental Algorithmics*, 10(2.6):1–27, 2005.
17. W. Paul and J. Simon. Decision trees and random access machines. In *ZUERICH: Proc. Symp. Logik und Algorithmik*, pages 331–340, 1980.